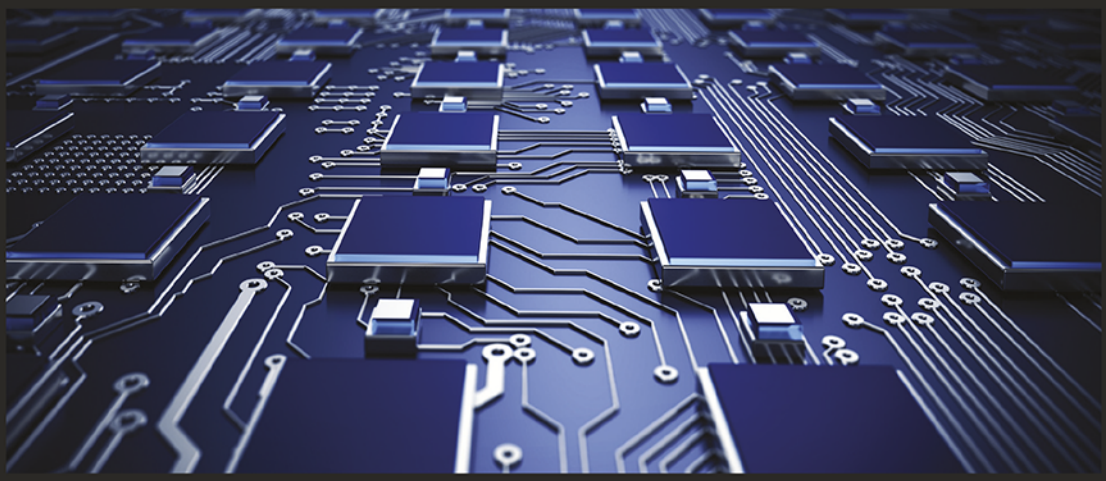
Helion

<packt>



Efektywne zarządzanie pamięcią w C++

Praktyczne strategie i techniki tworzenia lekkiego, bezpiecznego i niezawodnego oprogramowania

PATRICE ROY

Tytuł oryginału: C++ Memory Management: Write leaner and safer C++ code using proven memory-management techniques

Tłumaczenie: Robert Górczyński

ISBN: 978-83-289-3323-1

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/efzapa
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

CZĘŚĆ 3. Przejmowanie kontroli (nad mechanizmami zarządzania pamięcią)

DODATEK

Rozdział 2

Na co należy uważać?

Zdecydowałeś się przeczytać książkę o zarządzaniu pamięcią w C++ i jesteś gotowy zarówno na poznanie podejść wysokopoziomowych, jak i na „ubrudzenie rąk", aby uzyskać precyzyjną kontrolę nad procesem zarządzania pamięcią. Świetny plan!

Ponieważ będziesz tworzyć kod zarówno bardzo wysokiego, jak i bardzo niskiego poziomu, musimy upewnić się, że jesteś świadomy pewnych kwestii. Dzięki temu unikniesz problemów i nie będziesz tworzyć kodu, który wydaje się funkcjonować, ale w rzeczywistości nie działa, przynajmniej nie w sposób przenośny.

W tym rozdziale zwrócimy uwagę na niektóre aspekty programowania w C++, które będą istotne w całej książce, ale z którymi należy obchodzić się ostrożnie. Wprawdzie może to wyglądać jak (bardzo) krótkie kompendium złych praktyk lub zachęta do wpadania w tarapaty, ale potraktuj to raczej jako sposoby na dobre wykorzystanie nieco niebezpiecznych lub trudnych funkcji języka. Używasz C++, masz znaczną swobodę wyrazu i dostęp do funkcji, które są przydatne, jeśli dobrze je znasz i rozumiesz.

Chcemy tworzyć kod, który jest czysty i wydajny, oraz kształtować odpowiedzialnych programistów. Spróbujmy osiągnąć to razem.

W tym rozdziale omówimy następujące zagadnienia:

- Sposoby, w jakie można wpaść w kłopoty z kodem C++. Istnieją kwestie, których kompilator nie może wiarygodnie zdiagnozować, a także sytuacje, w przypadku których standard C++ nie określa, co się stanie. Tworzenie kodu, który robi takie rzeczy, to przepis na katastrofę — lub co najmniej zaskakujące czy nieprzenośne zachowanie.
- W szczególności przyjrzymy się, jak można doprowadzić do problemów podczas pracy ze wskaźnikami. Ponieważ ta książka omawia zarządzanie pamięcią, będziemy często używać wskaźników i związanej z nimi arytmetyki. Umiejętność rozróżnienia między właściwym i niewłaściwym ich użyciem będzie bardzo cenna.
- Na koniec omówimy, jakie rodzaje konwersji typów możemy wykonywać bez uciekania się do rzutowania (to będzie tematem następnego rozdziału) i dlaczego, wbrew powszechnemu przekonaniu, rzadko jest to dobry pomysł.

Naszym ogólnym celem będzie nauczenie się, czego nie powinniśmy robić (mimo że czasami będziemy wykonywać manewry, które to przypominają) i unikanie tych praktyk w przyszłości. Ponadto powinieneś zrozumieć, dlaczego postępujemy w taki sposób. W kolejnych rozdziałach skoncentrujemy się na tym, co *powinniśmy* robić i jak robić to dobrze!

Różne rodzaje zła

Zanim przejdziemy do omówienia konkretnych praktyk wymagających szczególnej uwagi, warto przyjrzeć się głównym kategoriom ryzyka, na jakie możemy się natknąć, jeśli tworzony przez nas kod nie przestrzega zasad języka. Z każdą taką kategorią wiąże się pewien rodzaj nieprzyjemności, których powinniśmy unikać.

Źle sformułowany kod, bez wymaganej diagnostyki

Niektóre konstrukcje w języku C++ określane są jako **źle sformułowane, bez wymaganej diagnostyki** (ang. *ill-formed, no diagnostic required*, IFNDR). W standardzie języka często można natrafić na sformułowania w stylu *„jeśli [...], program jest źle sformułowany, bez wymaganej diagnostyki*". Gdy coś jest IFNDR, oznacza to, że nasz program jest uszkodzony. Wprawdzie mogą wystąpić niepożądane skutki, ale kompilator nie jest zobowiązany do informowania nas o nich (czasami kompilator nie ma wystarczających informacji, aby zdiagnozować problematyczną sytuację).

Naruszenia **zasady jednej definicji** (ang. *one definition rule*, ODR), do których wrócimy w dalszej części tego rozdziału, zaliczają się do kategorii IFNDR. Istnieją jednak inne przypadki, takie jak posiadanie obiektu globalnego o różnych wymaganiach wyrównania (za pomocą `alignas`) w różnych jednostkach translacji (zasadniczo różnych plikach źródłowych) lub zdefiniowanie konstruktora, który deleguje do samego siebie bezpośrednio lub pośrednio. Oto przykład:

```
class X {
public:
    // #0 deleguje do #1, który deleguje do #0, który…
    X(float x) : X{ static_cast<int>(x) } { // #0
    }
    X(int n) : X{ n + 0.5f } { // #1
    }
};
int main() {}
```

Warto zauważyć, że kompilator może wyświetlić komunikat diagnostyczny, ale nie jest to wymagane. Nie chodzi o to, że kompilatory są leniwe — w niektórych przypadkach mogą nawet nie być w stanie dostarczyć informacji diagnostycznych! Dlatego należy uważać, aby nie tworzyć kodu, który prowadzi do sytuacji IFNDR.

Niezdefiniowane zachowanie

W poprzednim rozdziale wspomniałem o tzw. **niezdefiniowanym zachowaniu** (ang. *undefined behavior*, UB). Przez programistów C++ często jest ono postrzegane jako źródło problemów i bólu głowy, ale odnosi się do każdego zachowania, dla którego standard C++ nie nakłada żadnych wymagań. W praktyce oznacza to, że jeśli utworzysz kod, który zawiera niezdefiniowane zachowanie, wówczas nie masz pojęcia, co się stanie podczas wykonywania programu (przynajmniej jeśli dążysz do otrzymania względnie

przenośnego kodu). Klasyczne przykłady niezdefiniowanego zachowania obejmują dereferencję wskaźnika `null` lub niezainicjalizowanego — jeśli to zrobisz, będziesz mieć poważne problemy.

Z perspektywy kompilatorów niezdefiniowane zachowanie nie powinno występować (kod przestrzegający zasad języka nie prowadzi do UB). Z tego powodu kompilatory przeprowadzają „optymalizacje wokół" kodu zawierającego niezdefiniowane zachowanie, co może prowadzić do zaskakujących efektów: usunięcie testów i gałęzi, optymalizacja pętli itp.

Efekty niezdefiniowanego zachowania mają tendencję do lokalności. Na przykład w poniższym fragmencie kodu istnieje test sprawdzający, przed użyciem `*p` w jednym przypadku, czy `p` nie ma wartości `null`, ale istnieje co najmniej jeden dostęp do `*p`, który nie jest sprawdzany. Ten kod jest uszkodzony (niesprawdzony dostęp do `*p` to przykład niezdefiniowanego zachowania), więc kompilator może go przepisać w taki sposób, że wszystkie testy sprawdzające, czy `p` nie ma wartości `null`, zostaną usunięte. W końcu, gdyby `p` miało postać `nullptr`, to oznacza, że szkoda została już wyrządzona. Zatem kompilator ma prawo założyć, że programista przekazał do funkcji wskaźnik inny niż `null`!

```
int g(int);
int f(int *p) {
    if(p != nullptr)
        return g(*p); // w porządku, wiemy, że p ma wartość inną niż null
    return *p; // ups, jeżeli p == nullptr, wówczas mamy niezdefiniowane zachowanie
}
```

W tym przypadku cały kod funkcji `f()` mógłby zostać przepisany przez kompilator jako `return g(*p)`, zaś polecenie `return *p` stałoby się nieosiągalnym kodem.

Potencjalne niezdefiniowane zachowanie kryje się w różnych miejscach języka, w tym w przepełnieniu liczb całkowitych ze znakiem, dostępie do elementów tablicy poza jej granicami, wyścigach danych itd. Trwają ciągłe wysiłki, które mają na celu zmniejszenie liczby potencjalnych przypadków niezdefiniowanego zachowania (do tego zadania została nawet powołana specjalna grupa robocza **SG12**), ale prawdopodobnie pozostanie ono częścią języka w dającej się przewidzieć przyszłości i musimy być tego świadomi.

Zachowanie definiowane przez implementację

Niektóre fragmenty standardu podlegają tzw. **zachowaniu definiowanemu przez implementację**, czyli zachowaniu, na które można liczyć w przypadku konkretnej platformy. Jest to zachowanie, które Twoja platforma powinna udokumentować, ale które nie jest gwarantowane jako przenośne na inne platformy.

Zachowanie definiowane przez implementację występuje w wielu sytuacjach i obejmuje obszary takie jak limity definiowane przez implementację: maksymalna liczba zagnieżdżonych nawiasów, maksymalna liczba etykiet `case` w poleceniu `switch`, rzeczywista wielkość obiektu, maksymalna liczba wywołań rekurencyjnych w funkcji `constexpr`, liczba bitów w bajcie itd. Inne dobrze znane przypadki zachowania definiowanego przez implementację to liczba bajtów w obiekcie typu `int` lub to, czy typ `char` jest ze znakiem, czy bez znaku.

Wprawdzie zachowanie definiowane przez implementację samo w sobie nie jest źródłem problemów, ale może okazać się kłopotliwe, jeśli dąży się do otrzymania przenośnego kodu, a jednocześnie opiera się na pewnych nieprzenośnych założeniach. Czasami warto jawnie wyrazić swoje założenia w kodzie za pomocą `static_assert`, gdy założenie może być zweryfikowane w czasie kompilacji. Ewentualnie za pomocą podobnych mechanizmów, które działają w czasie wykonania, aby zdać sobie sprawę — zanim będzie za późno — że te założenia są błędne dla danej platformy docelowej.

Spójrz na przedstawiony tutaj kod funkcji `main()`:

```
int main() {
    // w kodzie przyjęto założenie, że wielkość typu int wynosi 4 bajty,
    // to nie jest przykład przenośnego założenia
    static_assert(sizeof(int)==4);
    // kod zostanie skompilowany tylko wtedy, gdy warunek jest prawdziwy...
}
```

O ile nie masz pewności, że Twój kod nigdy nie będzie musiał być przenoszony na inną platformę, staraj się jak najrzadziej polegać na zachowaniu, które pozostaje zależne od implementacji. Jeśli jednak musisz to zrobić, upewnij się o zweryfikowaniu takich sytuacji (najlepiej za pomocą wywołania `static_assert`, a jeśli to niemożliwe, to w czasie wykonania) i ich odpowiednim udokumentowaniu. Takie podejście pomoże uniknąć nieprzyjemnych niespodzianek w przyszłości.

Nieokreślone zachowanie (niezdefiniowane w dokumentacji)

Podczas gdy zachowanie zdefiniowane w implementacji jest nieprzenośne, ale udokumentowane dla danej platformy, zachowanie nieokreślone to takie, które nawet dla poprawnie napisanego programu z prawidłowymi danymi zależy od implementacji, ale nie musi być udokumentowane.

Niektóre przypadki nieokreślonego zachowania obejmują stan obiektu po przeniesieniu (określany jako *poprawny, ale nieokreślony*, więc bardziej chodzi o nieokreślony stan niż zachowanie), kolejność obliczania podwyrażeń w wywołaniu funkcji, czyli czy `f(g(),h())` najpierw obliczy `g()` czy `h()`, wartości w nowo przydzielonym fragmencie pamięci itp. Ten ostatni przykład jest interesujący dla naszych rozważań, ponieważ wersja robocza (debug) programu może wypełnić nowo przydzielone fragmenty pamięci rozpoznawalnym wzorem bitowym, aby pomóc w procesie debugowania. Z kolei zoptymalizowana wersja tego samego narzędzia może pozostawić początkowe bity nowo przydzielonego fragmentu pamięci „niezainicjalizowane", wraz z bitami, które miały w momencie alokacji, aby w ten sposób poprawić wydajność.

Zasada jednej definicji

Zasadę jednej definicji można przedstawić następująco: w jednostce translacji może istnieć tylko jedna definicja każdego „elementu" (funkcji, obiektu w zakresie, wyliczenia, szablonu itp.), chociaż może być wiele deklaracji tego elementu. Rozważmy następujący fragment kodu:

```
int f(int); // deklaracja
int f(int n); // to również jest deklaracja
int f(int m) { return m; } // definicja
// int f(int) { return 3; } // nie wolno tak zrobić (złamanie zasady jednej definicji)
```

W C++ unikanie naruszeń zasady jednej definicji jest bardzo ważne, ponieważ te „złośliwości" mogą umknąć uwadze kompilatora i wpaść w kategorię IFNDR. Na przykład ze względu na oddzielną kompilację plików źródłowych, jeżeli plik nagłówkowy zawiera definicję funkcji nieosadzonej, będzie ona powielana w każdym pliku źródłowym, który dołącza ten sam plik nagłówkowy. W efekcie każda kompilacja może zakończyć się sukcesem, zaś fakt istnienia wielu definicji tej samej funkcji w jednym projekcie może być odkryty dopiero później (na etapie linkowania) lub wręcz pozostać niewykryty, co z kolei spowoduje chaos.

Błędne zachowanie

Trwające prace związane z zapewnieniem bezpieczeństwa w C++ doprowadziły do dyskusji nad nowym rodzajem „złośliwości", która roboczo została nazwana *błędnym zachowaniem* (ang. *erroneous behavior*). Ta nowa kategoria ma obejmować sytuacje, które w przeszłości mogły być traktowane jako niezdefiniowane zachowanie, ale dla których możliwe byłoby przeprowadzenie diagnostyki i zapewnienie dobrze zdefiniowanego zachowania. Wprawdzie zachowanie to nadal byłoby niepoprawne, ale błędne zachowanie w pewien sposób ograniczałoby jego konsekwencje. Muszę w tym miejscu dodać, że w chwili powstawania książki prace nad koncepcją błędnego zachowania są w toku, a ta nowa funkcjonalność języka może pojawić się w standardzie C++26.

Jednym z przewidywanych zastosowań dla błędnego zachowania jest odczyt niezainicjalizowanej zmiennej, w przypadku którego implementacja mogłaby (ze względów bezpieczeństwa) zapewnić stałą wartość dla odczytywanych bitów. Twórcy implementacji będą zachęcani do diagnozowania błędu koncepcyjnego, który pojawia się na skutek odczytu takiej zmiennej. Innym przypadkiem zastosowania jest sytuacja, gdy programista zapomni o zwróceniu wartości z operatora przypisania dla typu innego niż `void`.

Skoro przyjrzeliśmy się już podstawowym „rodzinom" nieprzyjemności, które mogą dotknąć nasze programy, jeśli nie będziemy ostrożni, teraz możemy zagłębić się w wybrane z najważniejszych obszarów, które mogą przysporzyć nam kłopotów. Zobaczmy, czego powinniśmy unikać.

Wskaźniki

W poprzednim rozdziale omówiłem wskaźniki w C++ pod kątem tego, co reprezentują i co oznaczają. Ponadto wyjaśniłem, czym jest arytmetyka wskaźników i jakie daje nam możliwości. Teraz przyjrzymy się praktycznym zastosowaniom tej arytmetyki, zarówno właściwymi, jak i niewłaściwymi przypadkami użycia tego niskopoziomowego (ale czasami cennego) narzędzia.

Zastosowania arytmetyki wskaźników w tablicach

Arytmetyka wskaźników to przydatne, ale ostre narzędzie, którego łatwo nadużyć. W przypadku zwykłych tablic następujące dwie pętle, oznaczone jako *A* i *B*, działają dokładnie tak samo:

```
void f(int);
int main() {
   int vals[]{ 2,3,5,7,11 };
   enum { N = sizeof vals / sizeof vals[0] };
   for(int i = 0; i != N; ++i) //A
     f(vals[i]);
   for(int *p = vals; p != vals + N; ++p) //B
     f(*p);
}
```

Możesz się zastanawiać nad fragmentem `vals + N` w pętli *B* — to jest poprawny (i idiomatyczny) kod C++. Istnieje możliwość obserwowania wskaźnika tuż za końcem tablicy, mimo że nie wolno wówczas odczytywać wskazywanej przez niego wartości. Standard gwarantuje, że ten konkretny adres tuż za końcem tablicy jest dostępny dla programu. Jednak nie ma takiej gwarancji dla kolejnego adresu, więc zachowaj ostrożność!

Dopóki przestrzegasz zasad, możesz używać wskaźników do poruszania się w przód i w tył w obrębie tablicy. Jeśli jednak przekroczysz granice i użyjesz wskaźnika, aby wyjść poza jeden element za końcem tablicy, wkroczysz na teren niezdefiniowanego zachowania. Oznacza to, że możesz próbować uzyskać dostęp do adresu, który nie znajduje się w przestrzeni adresowej Twojego procesu.

```
int arr[10]{ }; // wszystkie elementy są zainicjalizowane z wartością zero
int *p = &arr[3];
p += 4; assert(p == &arr[7]);
--p;    assert(p == &arr[6]);
p += 4; // to wciąż akceptowalne, o ile nie spróbujesz uzyskać dostępu do *p
++p; // niezdefiniowane zachowanie, brak gwarancji działania zgodnie z oczekiwaniami
```

Wymienialność wskaźników

Standard C++ definiuje, co oznacza, że jeden obiekt jest **wymienialny wskaźnikowo** (ang. *pointer-interconvertible*) z innym. Ta wymienialność oznacza, że wskaźnika prowadzącego do pewnego obiektu można użyć jako wskaźnika do drugiego obiektu, zazwyczaj poprzez rzutowanie `reinterpret_cast` (dokładniej omówimy je w następnym rozdziale), ponieważ mają ten sam adres. Ogólnie rzecz biorąc, obowiązują następujące zasady:

- Obiekt jest wymienialny wskaźnikowo sam ze sobą.
- *Unia* jest wymienialna wskaźnikowo ze swoimi elementami składowymi danych, a także z ich pierwszymi elementami składowymi danych, jeśli są to typy złożone.
- Z pewnymi ograniczeniami `x` oraz `y` są wzajemnie wymienialne wskaźnikowo, jeśli jeden z nich jest obiektem, a drugi jest tego samego typu co pierwszy niestatyczny składnik danych tego obiektu.

W kolejnym fragmencie kodu znajduje się kilka przykładów:

```
struct X { int n; };
struct Y : X {};
union U { X x; short s; };
int main() {
    X x;
    Y y;
    U u;
    // element x jest wymienialny wskaźnikowo z x
    // element u jest wymienialny wskaźnikowo z u.x
    // element u jest wymienialny wskaźnikowo z u.s
    // element y jest wymienialny wskaźnikowo z y.x
}
```

Jeśli próbujesz użyć operatora `reinterpret_cast` w sposób, który nie przestrzega zasad konwersji wskaźników, Twój kod jest formalnie niepoprawny i nie ma gwarancji, że będzie działał w praktyce. Nie rób tego.

W zaprezentowanych przykładach kodu będziemy czasami korzystać z właściwości konwersji wskaźników. Pierwszy taki przykład znajduje się już w następnym punkcie.

Wykorzystanie arytmetyki wskaźników wewnątrz obiektu

Arytmetyka wskaźników wewnątrz obiektu jest również dozwolona w C++, choć należy zachować ostrożność przy jej stosowaniu (wykorzystaj odpowiednie rzutowania, które omówimy w następnym rozdziale, i upewnij się, że operacje arytmetyczne na wskaźnikach są wykonywane prawidłowo).

Na przykład wprawdzie poniższy kod jest poprawny, ale to nie postać, do której stosowania należy dążyć (nie ma sensu i wykonuje zadania w niepotrzebnie skomplikowany sposób, choć jest dozwolony i nie powoduje szkód):

```
struct A {
    int a;
    short s;
};
short * f(A &a) {
    // wymienialność wskaźnikowa w działaniu!
    int *p = reinterpret_cast<int*>(&a);
    p++;
    return reinterpret_cast<short*>(p); // dozwolone, choć w ramach
                                        // tego samego obiektu
}
int main() {
    A a;
    short *p = f(a);
    *p = 3; // poprawnie, z formalnego punktu widzenia
}
```

W tej książce nie będziemy nadużywać tego aspektu języka C++, ale musimy być świadomi jego istnienia, aby tworzyć poprawny kod niskiego poziomu.

Różnice między wskaźnikiem i adresem

W celu wzmocnienia bezpieczeństwa sprzętu i oprogramowania prowadzone są prace nad architekturami sprzętowymi, które mogą zapewnić formę „oznaczania wskaźników tagami", co z kolei umożliwiłoby między innymi śledzenie przez sprzęt źródła pochodzenia wskaźników. Dwa znane przykłady tego rodzaju architektury to CHERI (*https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.pdf?link_from_packtlink=yes*) oraz **Memory Tagging Extensions** — MTE (Linux: *https://docs.kernel.org/next/arch/arm64/memory-tagging-extension.html?link_from_packtlink=yes*, Android: *https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf* i *https://source.android.com/docs/security/test/memory-safety/arm-mte?link_from_packtlink=yes*, Windows: *https://raw.githubusercontent.com/microsoft/MSRC-Security-Research/master/papers/2020/Security%20analysis%20of%20memory%20tagging.pdf?link_from_packtlink=yes*).

Aby skorzystać z takich rozwiązań sprzętowych, język programowania musi rozróżniać między koncepcjami niskopoziomowych adresów i wysokopoziomowych wskaźników, ponieważ te drugie mogą uwzględniać fakt, że wskaźnik to coś więcej niż tylko lokalizacja w pamięci. Jeśli kod koniecznie musi porównywać niezwiązane ze sobą wskaźniki w celu ich uporządkowania, wówczas można je rzutować na `std::intptr_t` lub `std::uintptr_t` i porównywać wartości liczbowe zamiast faktycznych wskaźników. Warto zauważyć, że obsługa tych dwóch typów przez kompilator jest opcjonalna, choć wszystkie najważniejsze kompilatory oferują tę możliwość.

Wskaźnik null

Koncepcja wskaźnika pustego (ang. *null pointer*) jako rozpoznawalnej wartości dla wskaźników, które nie prowadzą do żadnej prawidłowej lokalizacji, sięga czasów C.A.R. Hoare'a. W języku C, poprzez makro `NULL`, ten wskaźnik był początkowo reprezentowany jako `char*` o wartości `0`, następnie jako `void*` o wartości `0`, natomiast w C++ po prostu jako wartość `0`, ponieważ konstrukcje takie jak `int *p = NULL;` z typowanym `NULL` były dozwolone w języku C, ale już nie w C++. Wynika to z bardziej rygorystycznego systemu typów w C++. Warto zauważyć, że wskaźnik o wartości `0` nie oznacza „wskazuj na adres zero", ponieważ ten adres sam w sobie jest całkowicie poprawny i jako taki jest używany na wielu platformach.

W C++ preferowanym sposobem wyrażania wskaźnika pustego jest `nullptr`, obiekt typu `std::nullptr_t`, który konwertuje się na wskaźniki dowolnego typu i zachowuje się zgodnie z oczekiwaniami. Rozwiązuje to w C++ wybrane długotrwałe problemy z literałem `0`, takie jak:

```
int f(int); // #0
int f(char*); // #1
int main() {
    int n = 3;
    char c;
    f(n); // wywołuje #0
    f(&c); // wywołuje #1
    f(0); // niejednoznaczne przed wprowadzeniem standardu C++11,
          // natomiast począwszy od C++11, wywołuje #0
    f(nullptr); // tylko począwszy od standardu C++11, jednoznacznie wywołuje #1
}
```

Należy pamiętać, że `nullptr` nie jest wskaźnikiem — to jest obiekt niejawnie konwertowany na wskaźnik. Z tego powodu cecha `std::is_pointer_v<nullptr>` zwraca wartość `false`, zaś C++ oferuje oddzielną cechę o nazwie `std::is_null_pointer<T>` przeznaczoną do statycznego sprawdzania, czy `T` jest typu `std::nullptr_t`, czy też nie (z uwzględnieniem `const` i `volatile`).

Odwoływanie się do wskaźnika pustego prowadzi do niezdefiniowanego zachowania, podobnie jak w przypadku wskaźnika niezainicjalizowanego. Celem używania `nullptr` w kodzie jest uczynienie tego stanu rozpoznawalnym: `nullptr` to wyraźnie określona wartość, podczas gdy niezainicjalizowany wskaźnik może mieć dowolną wartość.

W języku C++ (w przeciwieństwie do C) operacje arytmetyczne na wskaźniku pustym są dobrze zdefiniowane… o ile dodajemy do niego zero. Innymi słowy: dodanie zera do wskaźnika zerowego pozostaje w ramach poprawnego zachowania, ale dodanie czegokolwiek innego może prowadzić do nieprzewidywalnych efektów. Istnieje konkretny zapis dotyczący tego w standardzie C++ (*https://eel.is/c++draft/expr.add#4.1*). Oznacza to, że poniższy kod jest poprawny: w przypadku pustej tablicy wywołanie `begin()` zwraca `nullptr`, natomiast `size()` zwraca zero, więc funkcja `end()` w praktyce oblicza wartość `nullptr+0`, co jest zgodne z regułami.

```
template <class T> class Array {
    T *elems = nullptr; // wskaźnik do początku
    std::size_t nelems = 0; // liczba elementów
public:
    Array() = default; // = pusta tablica
    // ...
    auto size() const noexcept { return nelems; }
    // uwaga: wartością zwrotną może być nullptr
    auto begin() noexcept { return elems; }
    auto end() noexcept { return begin() + size(); }
};
```

Do tego przykładu z tablicami powrócimy bardziej szczegółowo w rozdziałach 12., 13. i 14. Pomoże on omówić kilka istotnych aspektów efektywnego zarządzania pamięcią. Na razie przyjrzyjmy się innemu źródłu ryzykownych operacji programistycznych.

Manipulowanie typami

Kolejnym obszarem, na którym programista C++ może wpaść w tarapaty, jest tzw. **manipulowanie typami** (ang. *type punning*). Przez manipulację typami rozumiemy techniki, które w pewnym stopniu obchodzą system typów języka. Uznanym narzędziem przeznaczonym do tego celu są operacje rzutowania, ponieważ pozostają one jawne w kodzie źródłowym i (poza rzutowaniem w stylu C) wyrażają intencję konwersji, ale ten temat zasługuje na osobny rozdział (następny, jeśli się nad tym zastanawiasz).

W tym podrozdziale przyjrzymy się innym sposobom manipulacji typami, zarówno tym zalecanym, jak i tym, których należy unikać.

Manipulacja typami poprzez elementy składowe unii

Unia to typ, w którym wszystkie elementy składowe znajdują się pod tym samym adresem. Wielkość unii jest równa wielkości jej największego elementu składowego, zaś wyrównanie unii jest najściślejszym wyrównaniem spośród jej elementów.

Rozważmy następujący fragment kodu:

```
struct X {
    char c[5]; short s;
} x;
// jeden bajt dopełnienia między x.c i x.s
static_assert(sizeof x.s == 2 && sizeof x == 8);
static_assert(alignof(x) == alignof(short));
union U {
    int n; X x;
} u;
static_assert(sizeof u == sizeof u.x);
static_assert(alignof(u) == alignof(u.n));
int main() {}
```

Kuszące może być przekonanie, że przedstawiona unia (konstrukcja `union`) pozwala przeprowadzać niejawną konwersję takich elementów jak czterobajtowa liczba zmiennoprzecinkowa na czterobajtową liczbę całkowitą. W języku C (ale nie C++) jest to rzeczywiście możliwe.

Mimo powszechnego przekonania, że taka praktyka jest dozwolona w C++, w rzeczywistości tak nie jest (z jednym szczególnym wyjątkiem, który omówimy za chwilę). W C++ ostatni element składowy unii, do którego coś zapisano, nazywany jest **aktywnym elementem składowym** unii i tylko z niego można w kodzie odczytywać dane. Dlatego poniższy kod jest nieprawidłowy — ponieważ odczytywanie z nieaktywnego elementu składowego unii prowadzi do niezdefiniowanego zachowania, które jest niedozwolone w przypadku funkcji `constexpr`:

```
union U {
    float f;
    int n;
};
constexpr int f() {
    U u{ 1.5f };
    return u.n; // niezdefiniowane zachowanie (u.f to aktywny element składowy unii)
}
int main() {
    // constexpr auto r0 = f(); // ten kod nie pozwala na kompilację
    auto r1 = f(); // ten kod się kompiluje, ponieważ nie użyto kontekstu
                   // constexpr, choć wciąż mamy niezdefiniowane zachowanie
}
```

Jak zapewne wiesz, funkcja typu `constexpr`, taka jak `f()` w poprzednim przykładzie, nie może zawierać kodu, który powodowałby niezdefiniowane zachowanie, jeśli zostałaby wywołana w kontekście `constexpr`. To czasami czyni ją interesującym narzędziem do zilustrowania pewnych kwestii.

Istnieje pewne zastrzeżenie dotyczące konwersji między elementami składowymi unii, które wiąże się z pojęciem wspólnej sekwencji początkowej.

Wspólna sekwencja początkowa

Jak wyjaśniono w dokumencie na stronie *https://eel.is/c++draft/class.mem.general#23*, **wspólna sekwencja początkowa** (ang. *common initial sequence*) dwóch struktur składa się z początkowych elementów składowych tych struktur, które mają odpowiadające sobie typy zgodne pod względem układu pamięci. Na przykład wspólna sekwencja początkowa struktur `A` i `B` składa się z ich dwóch pierwszych elementów (`int` jest zgodny układem z `const int`, zaś `float` jest zgodny z `volatile float`):

```
struct A { int n; float f; char c; };
struct B { const int b0; volatile float x; };
```

W przypadku unii możliwe jest odczytanie wartości z nieaktywnego elementu składowego, jeśli odczytywana wartość jest częścią wspólnej początkowej sekwencji elementów składowych (zarówno odczytywany, jak i aktywny). Spójrz na kolejny przykład:

```
struct A { int n0; char c0; };
struct B { int n1; char c1; float x; };
union U {
    A a;
    B b;
};
int f() {
    U u{ { 1, '2' } }; // inicjalizacja u.a
    return u.b.n1; // to nie jest niezdefiniowane zachowanie
}
int main() {
    return f(); // poprawne
}
```

Warto zauważyć, że takie zabiegi na typach powinny być stosowane z umiarem, ponieważ mogą utrudnić zrozumienie kodu źródłowego. Niemniej jednak mogą okazać się bardzo przydatne. Na przykład można je wykorzystać do implementacji ciekawych reprezentacji wewnętrznych dla klas, które mogą mieć dwie różne formy (takich jak `optional` czy `string`), co z kolei będzie ułatwiało przełączanie między nimi. Na tej podstawie można zbudować kilka użytecznych optymalizacji.

Typy intptr_t i uintptr_t

Jak wspomniano już wcześniej w rozdziale, w języku C++ nie można bezpośrednio porównywać wskaźników do dowolnych miejsc w pamięci w sposób dobrze zdefiniowany. Jednak w taki sposób można porównywać wartości całkowite, które są powiązane ze wskaźnikami. Spójrz na kolejny przykład:

```
#include <iostream>
#include <cstdint>
int main() {
    using namespace std;
    int m,
        n;
```

```
    // zwykłe porównywanie &m i &n jest niedozwolone
    if(reinterpret_cast<intptr_t>(&m) <
      reinterpret_cast<intptr_t>(&n))
      cout << "m precedes n in address order\n";
    else
      cout << "n precedes m in address order\n";
}
```

Typy `std::intptr_t` i `std::uintptr_t` są aliasami dla typów całkowitych, które z kolei są wystarczająco duże, aby pomieścić adres. W przypadku operacji, które mogą prowadzić do wartości ujemnych (np. odejmowanie), należy używać typu ze znakiem, `intptr_t`.

Funkcja std::memcpy()

Ze względów historycznych (i dla zachowania zgodności z językiem C) funkcja `std::memcpy()` ma szczególne właściwości, ponieważ może zainicjować cykl życia obiektu, jeśli zostanie odpowiednio użyta. Nieprawidłowe użycie `std::memcpy()` do manipulowania typem wyglądałoby następująco:

```
// załóżmy, że takie rozwiązanie sprawdza się w tym przykładzie
static_assert(sizeof(int) == sizeof(float));
#include <cassert>
#include <cstdlib>
#include <cstring>
int main() {
    float f = 1.5f;
    void *p = malloc(sizeof f);
    assert(p);
    int *q = std::memcpy(p, &f, sizeof f);
    int value = *q; // niezdefiniowane zachowanie
    //
}
```

Tego rodzaju rozwiązanie jest niedozwolone, ponieważ wywołanie `std::memcpy()` kopiuje obiekt typu `float` do obszaru pamięci wskazywanego przez `p`, co efektywnie rozpoczyna w tym miejscu cykl życiowy obiektu `float`. Ponieważ `q` jest wskaźnikiem do obiektu typu `int*`, próba odwołania się do niego prowadzi do niezdefiniowanego zachowania.

Z drugiej strony poniższy kod jest poprawny i pokazuje, jak można wykorzystać funkcję `std::memcpy()` na potrzeby manipulacji typem:

```
// załóżmy, że takie rozwiązanie sprawdza się w tym przykładzie
static_assert(sizeof(int) == sizeof(float));
#include <cassert>
#include <cstring>
int main() {
   float f = 1.5f;
   int value;
   std::memcpy(&value, &f, sizeof f); // ok
   //…
}
```

W tym drugim przykładzie użycie `std::memcpy()` do skopiowania bitów z `f` do `value` rozpoczyna cykl życiowy zmiennej `value`. Od tego momentu można jej używać jak każdej innej zmiennej typu `int`.

Szczególne przypadki char*, unsigned char* i std::byte*

Typy `char*`, `unsigned char*` (ale nie `signed char*`) oraz `std::byte*` mają w języku C++ status specjalny, ponieważ mogą dosłownie wskazywać dowolne miejsce w pamięci i tworzyć aliasy dla wszystkiego (*https://eel.is/c++draft/basic.lval#11*). Z tego powodu jeśli musisz uzyskać dostęp do bajtów reprezentujących wartość obiektu, wówczas te typy będą ważnym narzędziem w Twoim arsenale programistycznym.

W dalszej części książki będziemy czasami korzystać z tych typów podczas wykonywania niskopoziomowych operacji na bajtach. Należy pamiętać, że takie zabiegi są z natury niestabilne i nieprzenośne, ponieważ szczegóły takie jak kolejność bajtów w liczbie całkowitej mogą się różnić w zależności od platformy. Należy ostrożnie korzystać z takich niskopoziomowych narzędzi.

Funkcja std::start_lifetime_as<T>()

Ostatnimi narzędziami, które omówimy w rozdziale, są funkcje `std::start_lifetime_as<T>()` i `std::start_lifetime_as_array<T>()`. Wprawdzie były dyskutowane przez lata, ale zyskały na znaczeniu wraz z wydaniem standardu C++23. Ich rolą jest przyjmowanie jako argumentów czegoś takiego jak bufor nieprzetworzonych bajtów pamięci i zwracanie wskaźnika do obiektu typu `T` (wskazywanego przez ten bufor), którego cykl życiowy rozpoczął się. Dzięki temu od tego momentu można było używać obiektu, do którego prowadzi wskaźnik:

```
static_assert(sizeof(short) == 2);
#include <memory>
int main() {
    char buf[]{ 0x00, 0x01, 0x02, 0x03 };
    short* p = std::start_lifetime_as<short>(buf);
    // użycie *p jako skrótu
}
```

To jest funkcjonalność niskopoziomowa, której należy używać ostrożnie. Celem jest umożliwienie implementacji rozwiązań takich jak niskopoziomowe operacje wejścia/wyjścia na plikach czy kod przeznaczony do obsługi sieci (np. odbieranie pakietu UDP i traktowanie jego reprezentacji wartości, jakby był istniejącym obiektem) w czystym języku C++ bez wpadania w pułapkę zachowania niezdefiniowanego. Te funkcje bardziej szczegółowo omówimy w rozdziale 15.

Podsumowanie

W tym rozdziale omówiliśmy niektóre niskopoziomowe i czasem nieprzyjemne narzędzia, których niekiedy będziemy używać. Celem było zwrócenie uwagi na potencjalne zagrożenia i przypomnienie: musimy być odpowiedzialni oraz tworzyć rozsądny i poprawny kod, nawet mimo tego, że język programowania C++ daje pod tym względem dużą swobodę.

Podczas tworzenia w dalszych rozdziałach tej książki zaawansowanych mechanizmów zarządzania pamięcią te niebezpieczne narzędzia czasami okażą się przydatne. Miej na uwadze zamieszczone w tym rozdziale informacje dotyczące elementów wymagających ostrożności, korzystaj z tych narzędzi bardzo oszczędnie, ostrożnie oraz w sposób utrudniający ich niewłaściwe użycie.

W następnym rozdziale przyjrzymy się kluczowym operacjom rzutowania, które są dostępne w C++. Celem jest zrozumienie mechanizmów poszczególnych rodzajów rzutowania oraz tego, kiedy (i w jakim celu) należy go używać, abyśmy mogli następnie tworzyć potężne abstrakcje zarządzania pamięcią, które chcemy wykorzystywać.

Opanuj zaawansowane techniki zarządzania pamięcią w nowoczesnym C++

Zarządzanie pamięcią w C++, jeden z najbardziej wymagających aspektów tego języka programowania, stanowi jednocześnie klucz do tworzenia wydajnych i bezpiecznych aplikacji. W dobie rosnących wymagań dotyczących optymalizacji kodu, zwłaszcza w systemach czasu rzeczywistego, grach i aplikacjach wbudowanych, umiejętność efektywnego kontrolowania mechanizmów alokacji pamięci staje się szczególnie ważną kompetencją każdego programisty C++. Książka Patrice'a Roya, członka Komitetu Standardów ISO C++, przedstawia kompleksowe podejście do opanowania tych zaawansowanych technik.

Autor systematycznie prowadzi Czytelnika przez wszystkie aspekty zarządzania pamięcią — od podstawowych koncepcji cyklu życiowego obiektów, przez implementację inteligentnych wskaźników, aż po zaawansowane techniki optymalizacji. Książka łączy teorię z praktyką, prezentując konkretne implementacje detektorów wycieków pamięci, alokatorów opartych na pulach i kontenerów generycznych. Szczegółowo omawia zarówno tradycyjne podejścia, jak i najnowsze rozwiązania wprowadzone w standardach C++20 i C++23, w tym polimorficzne zasoby pamięci (PMR) i mechanizmy placement new.

W książce:

- implementacja własnych inteligentnych wskaźników i kontenerów
- techniki optymalizacji oparte na pulach pamięci
- praktyczne narzędzia diagnostyczne
- mechanizmy alokacji dla systemów o niskim opóźnieniu i aplikacji czasu rzeczywistego
- bezpieczne techniki manipulacji typami i obsługi wyjątków w kontekście zarządzania zasobami

Patrice Roy programuje zawodowo w C++ od ponad 30 lat. Od 1998 roku wykłada informatykę — specjalizuje się w systemach czasu rzeczywistego i programowaniu gier. Od 2014 roku jest aktywnym członkiem Komitetu Standaryzacyjnego ISO C++, współpracuje z takimi ekspertami jak Michael Wong i Bjarne Stroustrup przy rozwoju standardów języka.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-3323-1
9 788328 933231
Cena: 89,00 zł
packt